GCPersist: An Efficient GC-assisted Lazy Persistency Framework for Resilient Java Applications on NVM

Mingyu Wu[†‡], Haibo Chen[†‡], Hao Zhu[⋄], Binyu Zang[†‡], Haibing Guan[†‡]
† Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
‡ Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
⋄ National University of Defense Technology

Abstract

The emergence of non-volatile memory (NVM) has stimulated broad interests in building efficient and persistent systems and programming models. However, most prior work is built atop an *eager persistency model*, which mandates applications to persist their data as soon as possible and thus causes considerable overhead. Besides, prior work mainly focuses on native languages and overlooks the interactions with the managed runtime system in a high-level language. Such issues limit the scope of applications on NVM, especially for resilient applications that already have reliable but inefficient recovery mechanisms. This paper proposes *GCPersist*, an easy-to-use NVM programming framework atop a *lazy persistency model* to defer the persistency of user data for better performance, with the assistance of the garbage collection (GC) module in the managed runtime. GCPersist further provides differentiated persistency modes to reduce the runtime overhead. We have implemented *GCPersist* on the HotSpot JVM of OpenJDK and the evaluation results on Intel Optane DC persistent memory devices show that *GCPersist* performs well with resilient applications (like Spark) by reducing the recovery time by up to 3.26X while introducing only 1-6% runtime overhead during normal execution.

CCS Concepts • **Software and its engineering → Runtime environments**; • **Hardware → Non-volatile memory**;

This work is supported in part by the National Natural Science Foundation of China (No. 61672345, 61925206, 61802416), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Corresponding author: Binyu Zang (byzang@sjtu.edu.cn).

VEE '20, March 17, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7554-2/20/03…$15.00
https://doi.org/10.1145/3381052.3381318

Keywords Non-Volatile Memory, Lazy Persistency Model, Java Virtual Machine, Garbage Collection

ACM Reference Format:
Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, Haibing Guan. 2020. GCPersist: An Efficient GC-assisted Lazy Persistency Framework for Resilient Java Applications on NVM. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3381052.3381318

1 Introduction

Non-volatile memories (NVM) promises to bring a revolution on the memory hierarchy due to its non-volatility, byte addressability, and near-DRAM speed. The appealing characteristics of NVM have attracted wide interest in building programming models and systems to take full advantage of it. However, most prior work is mainly based on native languages [4, 9–11, 18, 27, 32, 37, 42]. There are some recent interests in extending NVM support from native code into high-level programming languages like Java [20, 36, 40]. However, they all provide similar *eager persistency models*: when write operations are issued, they are expected to be persisted as soon as possible. For example, Espresso [40] provides the *flush* APIs to flush cache lines into NVM right after they are written, while AutoPersist [36] guarantees that all objects reachable from a *durable root* are timely persisted by the Java runtime (JVM).

While an eager persistency model is suitable for applications requiring strict crash consistency, many applications only require a relaxed crash consistency such that data is not necessarily persisted right after it is written. A good example is big-data processing frameworks, which have generated a logical execution plan in advance for fault-tolerance. Such frameworks allow users to checkpoint intermediate data for faster recovery, but if the checkpoint is not ready, they can always turn to an alternative mechanism, where the lost data is recomputed according to the logical plan. For such applications, the eager persistency model may cause unnecessary but prohibitive overhead.

To this end, we propose *GCPersist*, a Java NVM framework that makes a novel synergy between lazy persistency and garbage collection (GC). The lazy persistency model in

GCPersist guarantees that all objects reachable from persistent roots, which are marked by applications, are automatically persisted after a GC cycle. Thanks to the reachability-based workflow of GC, the data copying process for persistency can be perfectly piggybacked with GC.

Nevertheless, since the access latency in NVM is still larger than DRAM, if the data is directly moved to NVM during GC, applications need to access their datasets at NVM and suffer from performance slowdown. To this end, *GCPersist* further provides two different persistency levels: *single-copy persistency* and *snapshot persistency*. In *single-copy persistency*, objects reachable from persistent roots only reside in NVM. Such objects will serve read/write operations regardless of crashes. In contrast, *snapshot persistency* will store a read-only *snapshot* in NVM while keeping objects still in DRAM. Objects in DRAM will serve the read/write operations from applications to reduce overhead without crashes, while those in NVM will only be used after a crash. Applications are free to choose between the two modes according to different scenarios.

GCPersist is implemented atop the garbage collector in Espresso [40], which is originally based on Parallel Scavenge Garbage Collector (PSGC), the default collector in OpenJDK 8. The evaluation atop Spark shows that *GCPersist* can reduce the recovery time by up to 3.26X while only introducing 1-6% runtime overhead, which is significantly better compared with prior work. Our evaluation suggests that *GCPersist* can enable more applications to benefit from NVM.

To summarize, the contributions of this paper is as follows:

- A novel lazy persistency model which exploits the interaction between applications and the garbage collector in managed runtime languages (Section 3).
- *GCPersist*, an easy-to-use NVM programming framework atop the lazy persistency model to enable fast recovery for resilient applications (Section 4).
- An evaluation on Spark to confirm that *GCPersist* recovers from failure notably faster with little overhead during normal execution (Section 5).

2 Background and Motivation

2.1 Programming with NVM in Java

High-speed, byte-addressable NVM has been commercially available as Intel releases its Optane DC Persistent Memory [19]. Similar to DRAM, NVM is also attached to the memory bus. However, since mainstream processors are equipped with volatile caches between CPUs and the memory subsystem, data will not become persistent until it leaves the cache. To guarantee persistency, Intel has introduced an instruction named *clwb* to explicitly write back a cache line into NVM. Since *clwb* can be executed in an out-of-order fashion, developers further need to insert fence instructions (*sfence*) to enforce the order. For example, Figure 1 shows the pseudo code to insert an element to the head of a list. After allocating a list from NVM (suppose the NVM allocator has ensured its persistency), both the data and the reference to the next element should be persisted via *clwb*. Since the data field should be persisted before the reference field to avoid pointing to corrupted data, we should further add a *sfence*.

```
1  function List insert(List l, Element e) {
2      List *new_list = new List();
3      new_list->data = e;
4      clwb(new_list->data);
5      sfence();
6      new_list->next = l;
7      clwb(new_list->next);
8      sfence();
9      return new_list;
10 }
```

Figure 1. Pseudo code of a function for inserting an element to a list.

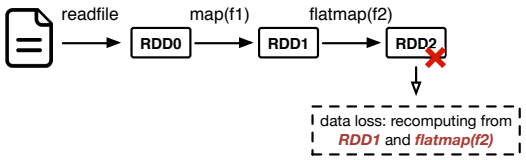


Figure 2. An example of an RDD chain and its corresponding recovery mechanism

Unfortunately, for high-level programming languages like Java, hardware instructions like *clwb* and *sfence* are not reachable for developers. Therefore, prior work like Espresso [40] and AutoPersist [36] has provided abstractions to connect Java programmers with NVM. Their programming models are illustrated in Figure 3.

Espresso (Figure 3a) proposes *pnew* to allocate memory from NVM for persistent objects. It further provides flush-related high-level APIs to ensure the persistency in field-level. *pnew* and flush APIs allow developers to manipulate persistent data in a fine-grained way. AutoPersist (Figure 3b) instead proposes an easy-to-use persistency framework, where developers only annotate some objects as *durable roots*. AutoPersist will automatically guarantee that all objects reachable from persistent roots will be copied to NVM and persisted. When the persistency order matters, AutoPersist requires applications to wrap their code into failure-atomic regions.

Although the programming model in Espresso and AutoPersist differ, they both provide *eager persistency*. Espresso encourages applications to invoke flush APIs to force cached

data to be written back, while AutoPersist guarantees that all write operations related to durable roots should be transformed to data manipulation in NVM, accompanied with *clwb* and *sfence* instructions. Therefore, they are mainly leveraged for applications with many fine-grained updates, such as key-value stores and databases.

2.2 Recovery in resilient applications

Large applications may have designed their own fault-tolerant protocols so that they can function well when encountering data loss. We name those applications *resilient applications*. Although the strict persistency model provided by Espresso and AutoPersist works well, it may hurt the performance of those applications due to a large amount of *sfence* and *clwb* instructions. In this work, we use Apache Spark [43], a state-of-the-art data processing framework, to exemplify those applications.

Spark enjoys fault-tolerant data processing mainly thanks to its data abstraction, Resilient Distributed Dataset (RDD). An RDD can be seen as a collection of data objects, which can be further partitioned and distributed into different machines. Each RDD is responsible for memorizing its *lineage*, i.e., how the data objects inside this RDD are computed from others. Even though data objects in an RDD are lost due to failures, they can be recomputed from other RDDs with the help of the lineage. Figure 2 illustrates an example of RDD chains in Spark. The chain starts with a file where serialized data is transformed into Java objects, and extends with various operators like *map* and *flatmap*. Suppose the data in RDD2 is lost due to failures like program crashes or machine shutdowns, the Spark worker will traverse backward on the chain, fetch data from RDD1, and recompute the lost data by reapplying the flatmap operator with the user-defined function *f2*. If the data in RDD1 is also unavailable, it should be recomputed by RDD0. For the worst case, the worker should re-read the file and recompute all RDDs on the chain.

Spark already provides a *cache* API for applications to explicitly keep some RDDs in memory for performance consideration. For example, machine learning applications may choose to store the training data in memory so as to process it in each iteration. However, cached RDDs are not resilient against crashes as they are stored in volatile memory. When a failure happens, Spark executors still need to recompute RDDs even though they have been cached, which leads to a costly recovery phase. The problem becomes more severe due to the fact that data processing applications like Spark are prone to out-of-memory (OOM) errors [13]. If another executor restarts by recomputing all the RDDs, it is likely to fail for the same reason. Therefore, it is natural to extend the semantic of *cache* to store RDDs in NVM for both close-to-DRAM access speed and fast recovery.

To support caching data in NVM, we have modified Spark to leverage the Persistent Java Heap (PJH) design in Espresso. When an RDD is going to be cached, Spark will store it into NVM with *pnew* and flush APIs. We have evaluated our modified Spark with a logistic regression application (provided by Spark itself), which trains a logistic regression model for several iterations on a cached training data set.

To understand the performance of recovery with NVM cache, we manually add some crash events by sending *kill* signals to the Spark executor. Before killing the executor, we also clean up the page cache to simulate a whole-system crash. When the executor is down, its manager (named *Worker* in Spark) will launch a new one for subsequent tasks. We mainly focus on the iteration where the crash happens, and leverage two metrics to measure the recovery efficiency: *overall execution time* for the iteration and *maximum task execution time* for tasks after the crash. The former metric shows the user-experienced latency of the iteration (in Spark, users can observe per-iteration execution time through GUI) while the latter one reflects the time occupied by recomputation due to a crash. Both of them can be collected from the log of Spark.

As illustrated in Figure 4a, caching RDDs in NVM significantly accelerates the recovery phase. Thanks to the NVM cache, the maximum task execution time compared to a DRAM one is reduced by 7.54X, while the overall execution time for the problematic iteration is reduced by 2.44X. The performance improvement mainly comes from reduced recomputation: with the NVM cache, the newly-launched executor can directly fetch cached objects without re-reading serialized file contents in the disk. We have also simulated a case where only the Spark executor is crashed due to software error by avoiding flushing the file cache (*DRAM+pagecache*) in Figure 4a). In this case, our NVM-cache version (with file cache flushed) still reduces the maximum task execution time and the overall execution time by 2.44X and 0.59X, which further shows the satisfying recovery efficiency with NVM.

Unfortunately, although storing objects in NVM enables fast recovery, the overhead for normal execution is prohibitive. When running the logistic regression algorithm for 100 iterations with crashes, the execution time with NVM cache increases from 112.976s to 145.754s (increased by 29.0%). To further understand the performance overhead, we present the execution time of the first ten iterations in Figure 4b. For the first iteration in the logistic regression application, Spark needs to read the file from disk and cache the contents in memory in the form of Java objects. Since our modified Spark requires allocating and persisting objects into NVM, the execution time is doubled. For later iterations, the average execution time for NVM cache is 21.1% larger, mainly

```
1  List insert(List l, Element e) {
2      List new_list = pnew List();
3      // Metadata-related operations
4      Field data = l.getClass().getDeclaredField("data");
5      Field next = l.getClass().getDeclaredField("next");
6      new_list.data = e;
7      data.flush(new_list.data);
8      new_list.next = l;
9      next.flush(new_list.next);
10 }
```

(a) Espresso

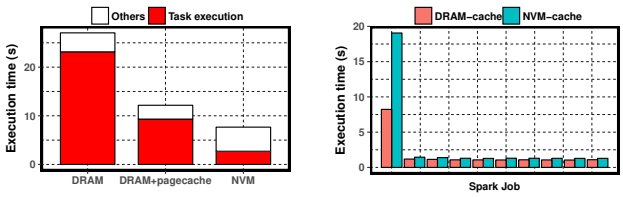
```
1  @durable_root
2  public static List global_list;
3  List insert(List l, Element e) {
4      atomic {
5          List new_list = new List();
6          new_list.data = e;
7          new_list.next = l;
8          global_list = new_list;
9      }
10 }
```

(b) AutoPersist

Figure 3. Prior NVM programming models in Java

due to the performance gap between DRAM and NVM[1], which is also considerable.

To summarize, leveraging NVM cache in resilient applications like Spark can dramatically improve the recovery efficiency, but it also results in considerable overhead to their normal executions mainly because of (1) eagerly flushing objects for persistency and (2) the performance gap between DRAM and NVM. The prohibitive overhead hinders resilient applications to leverage the virtues of non-volatile memory, and this motivates us to search for another technique to accomplish loose but high-performance persistency.



(a) Recovery time after a crash

(b) Normal execution time without crashes

Figure 4. Evaluation on the logistic regression application in Spark, with and without NVM cache

3 Design of GCPersist

To mitigate the overhead introduced by the strict persistency model in prior systems, we propose *GCPersist*, which instead integrates with the GC module in managed runtime to provide a *lazy persistency model* for better performance.

3.1 Lazy persistency model with runtime support

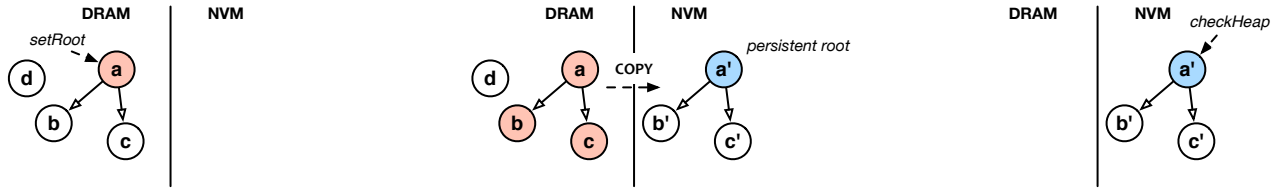
The core idea behind *GCPersist* is the lazy persistency model. Rather than eagerly writing back dirty cache lines into NVM, the lazy persistency level can defer the data persistency operations to the future. Prior work has proposed designs to relax the persistent model to mitigate the overhead introduced by expensive cache flush/write-back instructions. For example, Pelley et al. [35] propose epoch persistency, which exploits *persistency barriers* before which previous write operations must reach NVM. Nawab et al. [32] introduce *periodic persistency*, which assumes a global fence to write back all cached data to NVM. Those persistency models can improve the performance by reducing the number of cache write-back instructions, but they mainly have two limitations. First, they are mainly designed for implementing transactional data structures, where their models can be used to persist cached data when a transaction commits, but it is not clear how to generalize them to resilient applications like Spark. Second, they are designed to support applications in native code, and they do not consider high-level languages where applications are able to co-operate with the underlying runtime system. To this end, we decide to explore our own lazy persistency model with the managed runtime in mind.

Our persistency model assumes that the applications are running atop a managed runtime with Stop-The-World (STW) GC support, i.e., the runtime will periodically pause all application threads (or *mutators*) for memory reclamation. When mutators are paused, all of them reach a quiescent state where the referential integrity holds for all objects in the data heap. This assumption holds for many popular languages, such as Java, C#, Scala, etc. Our GC-assisted lazy persistency model is constructed by three components:

- *Persistent data identification.* Applications should inform the runtime which objects are required to be persisted through specific APIs.
- *Asynchronous reachability-based persistency.* When STW GC happens, the garbage collector will copy all objects supposed to be persistent to NVM.
- *Post-crash inspection.* Since the data persistency is achieved asynchronously, the managed runtime should provide APIs for applications to determine if the data objects have been persisted before crashes.

Figure 5 illustrates the three components of our lazy persistency model in *GCPersist*. We will discuss them respectively in the rest of this section.

[1]Although prior work assumes that the read latency of NVM is similar to DRAM, recent experiments [22] show that the latency for Intel Optane PM is more than 2x compared to that in DRAM.



(a) Persistent data identification (through *setRoot* API)

(b) Asynchronous reachability-based persistency with GC

(c) Post-crash inspection (through *checkHeap* API)

Figure 5. The components in the lazy persistency model of *GCPersist*

3.2 Persistent data identification

To persist data objects, applications should explicitly mark them so that the underlying runtime only moves them to NVM. However, persistent objects may appear at different locations in the source code, and manually annotating all of them may require considerable engineering work. Therefore, we provide a root-based API to simplify the annotation process.

A *root* is an object which is treated as "root of persistency". All objects reachable from a root (including itself) should be persisted into NVM. *GCPersist* introduces *setRoot(Object o, String name)* for applications to explicitly mark an object *o* as a persistent root and associate the object with a name, which is useful during recovery. The usage of *setRoot* is similar to the *@durable_root* annotation in AutoPersist [36]. *setRoot* significantly reduces the labor work required to mark persistent data. In Figure 5a, we mark object *a* with *setRoot* to instruct the managed runtime to persist both *b* and *c*. In the case of Spark, we only need to mark the array object in the RDD as a root so that the whole dataset can be persisted.

3.3 Asynchronous reachability-based persistency

GCPersist assumes that the runtime system periodically pauses all Java threads for GC. When GC begins, GC threads will start with known root objects, usually stack variables and global data structures of JVM (e.g., class-related metadata), to mark all reachable objects as alive. This marking process is very similar to our root-based persistency model, which also requires traversing from *persistent roots* to all reachable objects. Therefore, the persistency process can be perfectly integrated with GC.

In the design of *GCPersist*, GC will construct a *root set* to include both volatile roots (such as stack variables) and persistent roots (annotated by applications) so that GC threads will mark them together. Furthermore, GC threads should use two different marks, *alive* and *persistent*, to classify live objects. If an object is reachable from one which should be copied to NVM later, we will mark it as *persistent*. Otherwise, the object is only marked as *alive*.

When the mark phase ends, GC threads will continue to collect the heap. With the help of marks, GC threads can identify objects supposed to be persistent and thereby copy them to NVM. Note that *GCPersist* only requires the GC threads to mark live objects and is not bound with specific GC algorithms. Although the mark-based GC algorithm has many invariants (mark-sweep, mark-copy, mark-compact, etc.), *GCPersist* can be integrated with all of them. For mark-copy and mark-compact GC, since all marked objects should be copied to their corresponding new address, GC threads only need to modify the copy logic so that objects marked as *persistent* will be moved to NVM. For mark-sweep GC, GC threads need to scan the whole heap to clean up the dead objects. Therefore, when they find an object marked as *persistent*, they will copy it to NVM and put the original DRAM space into the free list. For objects marked as *alive*, GC threads should additionally inspect their references and modify those originally pointing to objects marked as *persistent* as they have been moved to NVM.

When GC ends, all objects reachable from persistent roots have been stored into NVM. Therefore, we insert a *fence* instruction to mark the end of the persistency process (similar to a global fence in [32]). After the fence instruction, all references in NVM will be safely pointed to other persistent objects, and no dangling pointers will be found even after a crash.

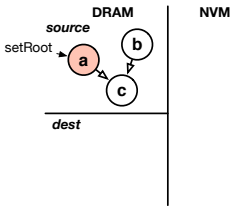
3.4 Post-crash inspection

Since *GCPersist* defers the persistency of objects to the GC point, the data can be lost when a crash happens before GC. Therefore, *GCPersist* should provide a mechanism so that those resilient applications can detect that their data has not been persisted in NVM and turn to their own fault-tolerance techniques. Besides, the APIs should also be simple to reduce the labor work in the recovery procedure.

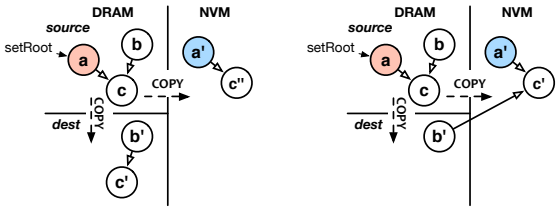
Objects in NVM are usually organized in a *heap* [4, 10, 21], where applications exploit *persistent roots* to access objects therein. Therefore, *GCPersist* provides *checkHeap* to check if all persistent objects can be safely accessed from roots. In the example of Figure 5c, the recovery threads will start from the persistent root *a'* to check if all reachable objects only contain references to persistent data.

Note that it is also possible to support finer-grained APIs like *checkRoot(Object o)* to check if objects reachable from

one specific root are persistent. However, since GC will ensure the persistency of a heap in an all-or-nothing fashion (through the global fence), *checkRoot* is not necessary.



(a) Before GC



(b) After GC (snapshot mode) **(c)** After GC (single-copy mode)

Figure 6. The heap image before/after GC in different persistency modes

3.5 Differentiated persistency modes

After *GCPersist* persists objects into NVM during GC, normal read/write operations from applications will be directly served by NVM. Unfortunately, since the basic performance of NVM is somewhat worse than DRAM, applications may still suffer from significant performance penalty (consider Figure 4). We provide differentiated persistency modes to solve this problem.

Since the read/write latency of NVM is still larger than DRAM, applications may want to access DRAM during normal execution and turn to NVM only when a crash happens. We thereby propose a mode named *snapshot persistency*, where persistent objects only serve as a read-only snapshot for those in DRAM. When GC threads find an object marked as *persistent*, rather than directly moving it into NVM, they will create two copies of it, move one copy to DRAM and the other to NVM. For references to those objects, GC threads should check if their owners are supposed to be persistent. If the references come from an object which also requires persistency, they should be modified to point to the copy in NVM, DRAM otherwise. For example (Figure 6a), suppose we have three live objects in DRAM required to be copied (suppose the managed runtime adopts a copy-based collector). Since object *c* is referenced by a persistent root *a* and a normal object *b*, the snapshot persistency mode in *GCPersist* requires it to be split into two copies: *c'* is moved to another place in DRAM while *c"* goes to NVM. As object *b* resides in DRAM, its reference will be modified to point to *c'*. In contrast, the reference in *a* will point to *c"*. When GC ends (Figure 6b), the objects in NVM are still self-contained, and *c"* serves as a read-only snapshot of *c'*. For object *a*, since no object in DRAM refers to it, it only keeps one copy *a'* in NVM as a persistent root. On the contrary, the original mode will only move *c* into NVM (shown in Figure 6c), so we refer it as *single-copy persistency mode*.

With the help of the snapshot persistency mode, applications can directly access cached objects in DRAM so that the runtime overhead without crashes is reduced. The snapshot mode can also be extended to other scenarios, such as snapshotting in NVM file systems [41]. Nevertheless, it also has some deficiencies:

Worse portability. Copy-based GC algorithms usually assume that an object only has one destination during GC, so they exploit *forwarding pointers* to store the destination for effective reference updates. Since the snapshot mode will assign two destinations for some objects, the forwarding pointer mechanism is no longer feasible. Those GC algorithms may require significant modifications to support the snapshot mode.

Larger memory footprint. Since the snapshot mode keeps two copies of the same object in DRAM and NVM respectively, the memory footprint is doubled. For scenarios where memory resource is scarce, the snapshot mode may not be suitable.

Due to those problems, although the snapshot mode is exploited by default in *GCPersist*, it can fall back into the single-copy mode in some cases. In our implementation (see Section 4), both modes have been implemented, and users can switch to the single-copy mode by configuring the JVM launch options.

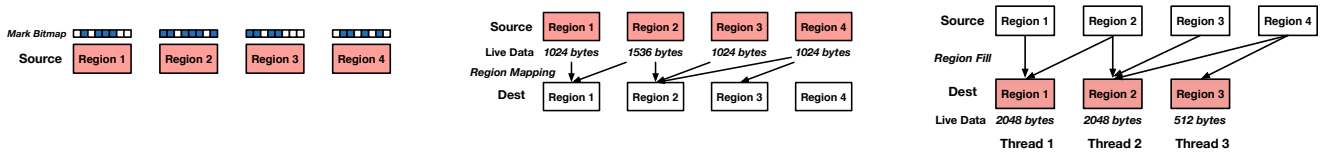
4 Implementation

4.1 Overview

To unlock lazy persistency for resilient applications, we have implemented *GCPersist* in the HotSpot JVM of OpenJDK 8. *GCPersist* inherits the heap structure in Espresso, which extends the traditional generational heap design to include Persistent Java Heap (PJH) for all persistent Java objects in NVM. Therefore, the heap layout in *GCPersist* consists of three parts: *young space* to store short-lived objects, *old space* to store long-lived objects, and PJH to store persistent objects. *GCPersist* is based on the Parallel Scavenge Garbage Collector (PSGC), the default GC in OpenJDK 8, to manage the heap space. It has carefully modified PSGC to automatically and efficiently persist objects into NVM.

4.2 Algorithm

PSGC provides two different algorithms to manage the heap. *Young GC* is used to collect only the young space, and objects in the old space will not be moved. In contrast, *Old GC*



(a) Mark phase: mark live objects in the mark bitmap

(b) Summary phase: count per-region live data and generate the region mapping

(c) Compact phase: fill regions in parallel according to the region mapping

Figure 7. The old GC algorithm in PSGC

is designed to scan the whole heap for memory reclamation. We have modified both two algorithms to support *GCPersist*.

4.2.1 Young GC

The young GC algorithm in PSGC is mark-copy: GC threads will mark all live objects in the young space and move them to their new destination. It seems to be a good fit for *GCPersist*, but *GCPersist* cannot be integrated with it because the young GC algorithm only collects the young space. If an object resides in the old space and entails persistency, the young GC algorithm will overlook it and result in crash inconsistency after GC. Therefore, *GCPersist* will not be activated during young GC, and all volatile objects reachable from NVM will still reside in DRAM.

Although those objects reachable from NVM will not be persisted during young GC, *GCPersist* should ensure that they are treated as live objects by GC threads. To achieve this, *GCPersist* adds all the persistent roots in PJH to the root set of young GC so that all objects reachable from those roots are scanned and marked. However, this technique is potentially inefficient as it requires to scan objects from persistent roots regardless of their location (DRAM or NVM), and we have provided optimizations for this problem (see Section 4.5).

4.2.2 Old GC

We integrate *GCPersist* with the old GC algorithm in PSGC to achieve lazy persistency. The algorithm is mark-compact: all live objects will be copied to the beginning of the heap to construct a large and contiguous free space. To support thread-level parallelism, PSGC has divided the heap into many *regions*. As illustrated in Figure 7, the algorithm contains three phases. In the first mark phase, GC threads will simultaneously mark all live objects and memorize them in a bitmap (named *mark_bitmap*). In the subsequent summary phase, live objects will be classified according to the region they reside so that each region can count the size of live objects. According to the per-region live data count, PSGC can generate a region mapping to specify which region the data objects will be copied after old GC. In the last compact phase, PSGC will decide which regions are required to be filled with live objects according to the region mapping. All regions will be abstracted as a *RegionTask* and pushed into a global work queue. GC threads will fetch tasks from the queue so that they can fill the corresponding regions with live objects in parallel. After copying live objects to the region, GC threads are also responsible for updating references therein.

All three phases in the old GC should be modified to support *GCPersist*. Algorithm 1 and 2 elaborate on the modified old GC algorithm in *GCPersist*. For simplicity, we skip the details on parallelism-related logic like task-fetching and only focus on four vital functions: *MarkObject*, *Summary*, *FillRegion*, and *RefUpdate*. We will first discuss the implementation of single-copy mode in that it is simpler, and describe the snapshot mode later.

In the mark phase, GC threads will leverage a stack to mark live objects. The marking algorithm starts with a reference to some root object (Line 2), either volatile or persistent, and continues with repeatedly fetching unprocessed references until the stack becomes empty. In addition to the *mark_bitmap*, we further add a new bitmap named *perm_bitmap* to mark objects reachable from persistent roots. When a reference is popped from the stack (Line 4), we will get the object *obj* it refers to (Line 5) and *referent* which holds this reference (Line 6). Afterward, we will determine if *obj* is reachable from NVM by checking if *referent* has been marked in our added *perm_bitmap* (Line 10). If *referent* is reachable from NVM, *obj* is also transitively reachable, so we will also mark it in *perm_bitmap*. Otherwise, *obj* will only be marked in the original *mark_bitmap*. After processing *obj* itself, we should also push its references into the working stack for further marking (Line 18-20).

In the summary phase, *GCPersist* will scan the region to count the size of live data. The idea is to find all live objects within the region from two bitmaps and add their sizes together (Line 26-37). Since *GCPersist* requires copying data to NVM, it maintains two counts: *mark_count* for live data in DRAM and *perm_count* for NVM. Note that if an object is in both two bitmaps, its size will only be added into *perm_count* as it will later be copied into NVM. According to those two numbers, *GCPersist* can calculate the destination region for the under-processed region (Line 38-46). A source region will have two kinds of destination regions: *pdest* in NVM and *dest* in DRAM. Source and destination regions keep references to each other to generate a region mapping.

Algorithm 1 The mark and summary algorithm for *GCPersist*

```
1: function MarkObject(root)
2:     workstack.push(root)
3:     while workstack.size() != 0 do
4:         ref ← workstack.pop()
5:         obj ← dereference(ref)
6:         referent ← GetOwner(ref)
7:         if isMarked(obj) then
8:             continue
9:         end if
10:        if perm_bitmap.isMarked(referent) then
11:            perm_bitmap.mark(obj)
12:            if SnapShotMode then
13:                mark_bitmap.mark(obj)
14:            end if
15:        else
16:            mark_bitmap.mark(obj)
17:        end if
18:        for ref in obj do
19:            workstack.push(ref)
20:        end for
21:    end while
22: end function
23: function Summary(region)
24:    nextobj ← Min(mark_bitmap.next(region.begin),
25:                  perm_bitmap.next(region.begin))
26:    while nextobj < region.end do
27:        if perm_bitmap.isMarked(nextobj) then
28:            region.perm_count += nextobj.size
29:            if SnapShotMode then
30:                region.mark_count += nextobj.size
31:            end if
32:        else if mark_bitmap.isMarked(obj) then
33:            region.mark_count += nextobj.size
34:        end if
35:        nextobj ← Min(mark_bitmap.next(nextobj),
36:                      perm_bitmap.next(nextobj))
37:    end while
38:    if region.perm_count != 0 then
39:        perm_region ← Calc(region, perm_count)
40:        perm_region.source ← region
41:        region.pdest ← perm_region
42:    else if region.mark_count != 0 then
43:        dest_region ← Calc(region, mark_count)
44:        dest_region.source ← region
45:        region.dest ← dest_region
46:    end if
47: end function
```

In the final compact phase, GC threads will fetch destination regions and fill them up with live objects from source regions. As Algorithm 2 shows, for a destination region, GC threads first find its source region (Line 2) and determine if it resides in NVM (Line 3). If the destination region is in NVM, GC threads will find all live objects within the source region by querying the *perm_bitmap* and move them to their new address (Line 6, 12). Otherwise, GC threads will turn to *mark_bitmap* and copy objects from DRAM (omitted in the algorithm).

After copying an object, GC threads are also responsible for updating references therein (Line 8-10). Since reference updates only happen after copying, all references should originally point to the address of an object before copying, i.e., its address in the source region. Therefore, to process a reference, GC threads first get the referred object *src_obj* and the source region it resides (Line 21-22). The next step is to determine if the reference is reachable from NVM by querying the *perm_bitmap* with the address of *src_obj*. If it is supposed to be copied into NVM, GC threads will generate *pdest_obj*, which is calculated from the source region's NVM destination *pdest* and *src_obj* (Line 23-25). The calculation algorithm is complicated and irrelevant, so we will not discuss the detail. *pdest_obj* will be assigned to the reference *ref* as the new address of *src_obj* (Line 30-31). If *src_obj* is not reachable from NVM, GC threads will generate *dest_obj* with a similar algorithm and store it to the reference (Line 26-27, 32-34).

The algorithm above is capable of supporting the *single-copy persistency mode*. As for the *snapshot persistency mode*, the logic is somewhat different. For the mark phase, if an object is reachable from NVM, as it will be split into two copies, we mark it in both two bitmaps (Line 13). Besides, both counts in the summary phase should consider the size of those objects (Line 30). In the reference updating algorithm, the new value of a reference depends on the location the reference resides. If the reference is in NVM, the objects it refers to must also be copied into NVM, so the assigned value will be *pdest_obj* (Line 37). If it is in DRAM, even though the object it refers to should be copied to NVM, its value should be modified to the destination address in DRAM *dest_obj* (Line 39). This difference highlights the characteristic of the snapshot mode: when old GC completes, no references in DRAM will point to NVM. Objects in NVM will be stored in the background and only become useful during crash recovery.

4.3 Programming APIs

Both *setRoot* and *checkHeap* required by the lazy persistency model are supported in our implementation. *setRoot* has already been implemented by Espresso: it leverages APIs to mark all persistent roots so that applications can access their persistent data after the PJH is reloaded. Therefore, we only need to implement *checkHeap* on our own.

Algorithm 2 The compact algorithm for *GCPersist*

```
1:  function FillRegion(region)
2:      src_region ← region.source
3:      if InNVM(region) then
4:          next_obj ← perm_bitmap.next(src_region.begin)
5:          while next_obj < src_region.end do
6:              Copy(region.top, next_obj)
7:              # region.top stores the data from next_obj
8:              for ref in region.top do
9:                  UpdateRef(region.top)
10:             end for
11:             region.top += next_obj.size
12:             next_obj ← perm_bitmap.next(next_obj)
13:         end while
14:     else
15:         # this branch has nearly the same logic except
16:           that it uses mark_bitmap
17:         .....
18:     end if
19: end function
20: function UpdateRef(ref)
21:     src_obj ← dereference(ref)
22:     src_region ← Locate(src_obj)
23:     if perm_bitmap.isMarked(src_obj) then
24:         pdest_obj ← CalcRef(src_region.pdest,
25:                             src_obj)
26:     else if mark_bitmap.isMarked(src_obj) then
27:         dest_obj ← CalcRef(src_region.dest, src_obj)
28:     end if
29:     if not SnapShotMode then
30:         if pdest_obj != NULL then
31:             ref ← pdest_obj
32:         else
33:             ref ← dest_obj
34:         end if
35:     else
36:         if InNVM(ref) then
37:             ref ← pdest_obj
38:         else
39:             ref ← dest_obj
40:         end if
41:     end if
42: end function
```

A straw-man implementation of *checkHeap* would be traversing the persistent heap through all roots and check if there are dangling references out of NVM. However, since our implementation of GC has guaranteed that all references remain intact when old GC ends, traversing the heap after a crash is no longer necessary. We identify the validity of the whole heap by exploiting a byte in the metadata part of the PJH, which also resides in NVM together with persistent objects. When the object content in NVM is modified by applications or GC threads, we will mark the heap as invalid, and it is unrecoverable. When GC ends, the whole heap becomes valid and recoverable. When a crash occurs, *checkHeap* simply returns the value stored in the validity byte. This solution avoids prohibitive heap traversal and accelerates the recovery time.

4.4 Persistency Guarantee

Stabilizing persistency points. *GCPersist* relies on the old GC to guarantee the persistency of user data. However, the frequency of old GC is unstable and highly dependent on the applications and JVM configurations. When the memory resource is abundant, or the young GC algorithm is quite effective, old GC rarely happens. Therefore, we have modified the policy in JVM to determine whether old GC is required. As mentioned above, when *setRoot* is invoked, or an object in NVM is modified, we will mark the heap as invalid. When young GC happens, *GCPersist* will check the validity byte and maintain a counter to memorize how many young GC cycles it has witnessed after the persistent heap becomes invalid. When the counter reaches a threshold, an old GC phase will be triggered, so the persistent heap becomes valid again. We have proposed a JVM option *GCPersistThreshold* to control the number of young GC the application can tolerate without a valid persistent heap. We also provide a timer-based policy: when the time duration after the heap becomes unrecoverable exceeds a preset threshold (e.g., a second), *GCPersist* will force old GC to happen.

Deferred persistency. According to the lazy persistency model, writes to NVM are not necessarily persistent until the global fence. Therefore, we will only issue batched cache flushing operations (*clwb*) at the end of old GC. Since we are still in the Stop-The-World environment, *GCPersist* can leverage all GC threads to simultaneously flush the cache for better performance. Afterward, we only insert one *fence* instruction to serve as the global fence. Due to the parallel cache flushing and single fence instruction, the performance of *GCPersist* is much better than prior work.

4.5 Optimizations on read-mostly datasets

Although the persistent heap turns unrecoverable after a single write to NVM, directly writing to NVM happens very rarely in many scenarios. In big-data processing applications like Spark, the data abstraction (RDD) is immutable and cannot be directly modified by applications. Therefore, once the data has been persisted in NVM, they will only serve read operations. In the snapshot use cases like file systems and databases, the snapshot will be written-once and read-only. Consequently, those datasets stored in NVM are read-mostly. According to the observation, we have proposed several optimization techniques to further improve the performance of *GCPersist*.

Detecting writes to NVM. Since write operations on NVM will invalidate the persistent heap, *GCPersist* must accurately detect them. A traditional solution to this problem is *write barriers*, with which JVM instruments detection code before each write operations from Java applications. However, this solution will affect the performance of each write and thus slow down the applications. We instead exploit a hardware-based solution. When the persistent heap is created, or the old GC has validated the heap, we will mark the whole heap as read-only in the page table, with *mprotect* calls. Therefore, every time when a write hits on NVM, a page fault will be triggered, and JVM can detect it through a pre-registered page fault handler and thereby set the validity byte. Once the heap becomes invalid, we invoke *mprotect* again to enable subsequent writes to all the memory pages in NVM. Although the overhead of page fault handling is much larger than a single write barrier, it happens only once while barriers may be executed millions of times.

Improving young GC. Since the young GC algorithm only collects the young space, it must ensure that all objects in both the old space and the persistent heap are alive and leverage them as roots. In our implementation, all persistent roots will be added into the root set of young GC for its mark-copy collection, which may induce unnecessary scans on the persistent heap. Fortunately, this problem can also be resolved with the memory protection technique. After the old GC ends, objects in NVM are self-contained, and no reference will escape from NVM. Therefore, when a new young GC cycle is triggered, *GCPersist* will check if the persistent heap still remains valid. If the heap is still valid, its contents are not changed, and no reference in NVM will point to DRAM. Consequently, *GCPersist* can avoid adding persistent roots into the root set, and the young GC will become as effective as one without using NVM.

Detecting append operations. As analyzed above, when a dataset has been copied into NVM, it is unlikely to be updated. Therefore, writes that invalidate the persistent heap are usually *setRoot* calls to *append* new datasets into NVM. *GCPersist* tries to leverage this behavior by differentiating the allocated memory from the unallocated. When a page fault occurs, *GCPersist* will check the faulted address to determine whether it is an *append* request to expand the persistent heap or an *update* request to modified the allocated contents. If it is an *append* operation, *GCPersist* will only enable writes to the unallocated areas.

This technique distinguishes those *active* datasets which have not been copied into NVM from others. It would be useful for the continuous checkpoint case, where datasets are iteratively generated and checkpointed for fast recovery. When the JVM crashes, objects in the old checkpoints are still treated as valid and thus can be reused for error recovery.

5 Evaluation

5.1 Experiment setup

GCPersist is implemented atop Espresso JVM, which is a modified version of OpenJDK 8u102-b14. We provide a JVM option *PersistByGC* so that applications are free to disable *GCPersist* if they want to manage the persistent data on their own. It takes about 1,800 LoCs to implement *GCPersist*.

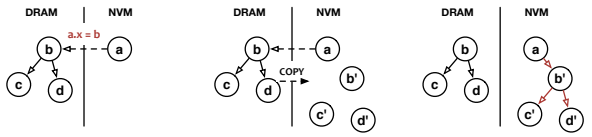
We leverage Spark, a well-known big-data processing framework, as an example of resilient applications. The version of Spark is 2.1.0. To support caching objects in NVM for Spark, we mainly modify three parts. First, since Spark provides a *persist* interface for users to specify which level of storage the RDD should be cached (DRAM, disk, or both), we add a new level named *NVM*. Second, we modify the *MemoryStore* class to create persistent heap and identify RDDs cached in NVM. When it finds that an RDD should be stored to NVM (specified by applications), it will invoke *setRoot* to connect the data array with a reference in NVM. We also modify the scheduler to avoid recomputation when data in NVM is available. Thanks to the easy-to-use APIs of *GCPersist*, we only need about 50 LoCs to implement the core part of the recoverable NVM cache for Spark (By comparison, the codebase of Spark is over 140K LoCs).

Four applications are exploited to evaluate our *GCPersist*: *PageRank* (PR), *KMeans* (KM), *LogisticRegression* (LR), and *TransitiveClosure* (TC). All of them can be found in the example folder of Spark. For LR and KM we leverage synthesized datasets (1.4G) for training, while for PR and TC we use two real-world graphs respectively: Berkeley/Stanford [24] (15K vertices, 171K edges) and Blogs [1] (1K vertices, 19K edges). For *GCPersist*, all those datasets will be cached into NVM for later reuse. Although the datasets for PR and TC are not large, they will be quickly inflated and induce a large memory footprint during runtime as they introduce many shuffle operations.

Due to the limited number of available NVM machines, all experiments are conducted on one server with dual Intel Xeon Gold 5125M CPUs (20 logical cores for each). Each CPU is equipped with six Intel Optane DC PM devices (128GB for each PM device, 1.5T altogether). To avoid NUMA issues, we allocate persistent memory from a single CPU, and bind all threads of Spark executor onto it to force local memory access. The heap size for the Java heap is 20GB while the size of PJH is 2GB. Since Spark only caches one replica for each RDD, the recovery time will not be affected much with a multi-node setting.

5.2 Baseline

We mainly compare our *GCPersist* with two prior projects: *Espresso* and *AutoPersist*. We also leverage a vanilla Spark running on the Espresso JVM (without using NVM) as an ideal baseline. Unfortunately, since AutoPersist is built on the Maxine JVM while Espresso and *GCPersist* are built on



(a) Application issues *a.x=b* to activate AutoPersist

(b) AutoPersist transitively copies objects to DRAM

(c) The reference will be updated after all objects have been persisted

Figure 8. The workflow of AutoPersist

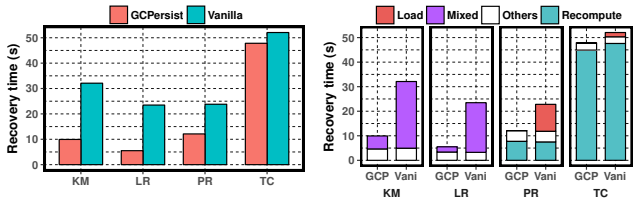
the HotSpot JVM, it would be unfair to directly compare their performance. Therefore, we have implemented a simplified version of AutoPersist on the HotSpot VM, which we refer to as *AutoPersist** in our evaluation.

AutoPersist is an easy-to-use Java NVM programming framework which also leverages runtime to automatically persists Java objects into NVM. In AutoPersist, applications are asked to annotate *durable roots*, and the underlying runtime ensures that all objects reachable from roots are persisted in NVM as soon as possible. When an application thread creates a reference from NVM to DRAM, the runtime must detect it and transitively persist all objects reachable from NVM. As Figure 8 illustrates, when the application issues *a.x = b* to connect object *a* in NVM with *b* in DRAM, AutoPersist will automatically calculate a *transitive closure*, which contains all DRAM objects reachable from *a*. For each object, AutoPersist ensures that it will not be persisted until all objects it refers has reached NVM. After all objects have been persisted, AutoPersist will finally update the reference in *a* to the new address of *b*, i.e., *b'*, to guarantee that no reference is out of NVM.

Implementing a full-fledged AutoPersist framework requires significant modifications to the JVM, including object layout, read/write barriers, and garbage collection. Therefore, we implement a simplified version named *AutoPersist** which only works in Spark. When Spark is going to cache an RDD in NVM, it will set the corresponding data array as a *durable root*, where we insert a JVM call named *NVMCopy* to transitively copy all objects reachable from the array into NVM. The implementation of *NVMCopy* mainly follows the transitive persistency algorithm in the paper of AutoPersist [36].

5.3 Recovery time

For recovery, we compare *GCPersist* with the vanilla Spark atop Espresso JVM where no NVM is used (Vanilla). We manually add *SIGKILL* signals in the same job for both and drop the file cache to simulate a machine crash. The recovery time is measured as the completion time of the first iteration after the crash. Figure 9a shows that the recovery time can be reduced for all four applications. The reduction is from 9% in TC to 3.26X in LR. Figure 9b further breaks the recovery time into three parts: data loading, recomputation, and others (including fault detection and executor re-launching). Since the data loading and recomputation are overlapped in KM and LR, we merge them together in Figure 9b. The result shows that our *GCPersist* can help improve both data loading and recomputation thanks to caching data into NVM. The improvement for *GCPersist* drops for PR and TC as they are more computing-intensive and exploit smaller datasets. The result suggests that *GCPersist* can achieve even better recovery performance with larger datasets.



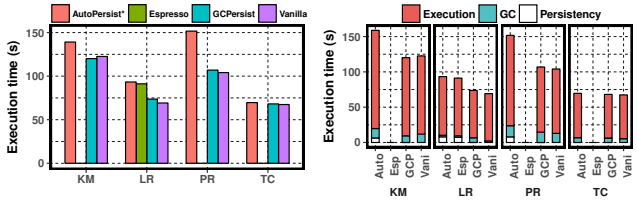
(a) Recovery time

(b) Breakdown

Figure 9. The crash recovery time of *GCPersist* against the vanilla baseline with DRAM cache

5.4 Normal execution time

For normal execution time, we compare our *GCPersist* against Espresso (with *pnew*) and AutoPersist*. We still use the Espresso JVM with DRAM cache (Vanilla) as the best result *GCPersist* can achieve. Figure 10 shows the execution time for three applications. Since the cached data entries may contain Scala objects, which is not supported by *pnew*, we only provide the evaluation result in LR for Espresso.



(a) Normal Execution Time

(b) Breakdown

Figure 10. The normal execution time without crash for *GCPersist* and prior work

As Figure 10 illustrates, the performance of our *GCPersist* is close to the vanilla baseline and introduces 1.01% - 6.49% overhead. For the KM case, the execution time for *GCPersist* is even better than the baseline. As shown in Figure 10b, *GCPersist* reaches similar execution time and slightly larger GC time compared with baseline. In contrast, the normal execution time dramatically increases when applications exploit Espresso or AutoPersist* to persist their data into NVM. As for the LR case, the runtime overhead for Espresso and

AutoPersist* is 31.1% and 34.8%, respectively. Figure 10b further reveals that the overhead comes from eagerly flushing objects into NVM (*persistency*) and the performance gap between DRAM and NVM devices (*execution*).

Note that for the LR application, the performance for AutoPersist* is close to Espresso. Although AutoPersist* can reduce the number of cache write-back instructions as it has full knowledge of the object layout, it has two major problems. First, AutoPersist* leverages a queue to process references in a breadth-first-search fashion, which is known to squander the locality of applications. Second, AutoPersist* has to remember all the objects and references it has modified in a queue, and the size of the queue would be very large when the number of objects to be processed is huge. In LR, AutoPersist* needs to maintain an 80MB queue to remember objects. Besides those sources of overhead, the original AutoPersist also induces constant overhead (such as read-/write barriers), which has been hidden in our simplified implementation. The result suggests that both AutoPersist and Espresso is not suitable for resilient applications due to their over-constrained persistency models.

We have also evaluated the single-copy mode of *GCPersist* atop LR. Compared with the snapshot mode, the execution time is 8.2% larger because of the access overhead on NVM, but the DRAM footprint is reduced by 29.4%. This experiment suggests that the single-copy mode will be cost-effective in a memory-hungry environment.

6 Related Work

6.1 Heap-based data management in NVM

Many systems have been implemented to manage persistent data in NVM, and a well-known strategy is to manage NVM in heaps, where applications access persistent data through roots. NV-heaps [10] proposes a heap abstraction for flexible data management, high-performance data access, and referential integrity. Makalu [4] provides a crash-consistent heap allocator for NVM. Intel's persistent memory development kit (PMDK) [21] also supports heap-based memory management. Prior work like PCJ [20] and MDS [17] provides a shim layer in Java so that Java applications can access data in NVM. Espresso [40] and AutoPersist [36] directly manage Java objects in NVM to improve the access performance, and *GCPersist* has further extended them to support resilient applications with trivial overhead.

Another line of work enables data management in NVM for Java but for other considerations. Write-rationing GC [2, 3] migrates read-mostly Java objects to NVM to extend NVM lifetimes. Panthera [38] enables object management on hybrid memories for Spark to leverage the large capacity and energy-efficiency of NVM. In contrast, *GCPersist* focuses on the persistency characteristic of NVM to support fast crash recovery.

6.2 Lazy persistency model and systems

Compared with strict/eager persistency, lazy persistency model defers the data persistency for better performance. Pelley et al. [23, 35] propose epoch persistency to divide executions into epochs within which the persistency order does not matter. Nawab et al. [32] present periodic persistency where a global fence is periodically executed to write back all dirty cache lines into NVM. Our model instead includes the co-operation with garbage collector for better performance.

To avoid expensive cache flush and fence instructions, prior systems also leverage lazy persistency to asynchronously persist data into NVM. SoupFS [12] exploits the traditional soft update mechanism to move the cache flush instructions in NVM file systems off the critical path. DudeTM [27] and Kamino-Tx [30] delays the persistency of transactions by priorly writing to backup storage in DRAM. Pisces [15] avoids eagerly updating NVM data by leveraging dual-versioned concurrency control (DVCC) and loosening the serializability level to snapshot isolation (SI). *GCPersist* achieves lazy persistency by relying on the old GC to copy data into NVM.

6.3 Runtime optimizations for big-data processing frameworks

Big-data processing frameworks [8, 14, 16, 43] have grown up into an important kind of workload in managed-runtime languages. The rise of those frameworks has stimulated researches on developing a better managed runtime for them. Gerenuk [31] transforms the Java code to directly access native data for performance improvement. Yak [34] designs a big-data-friendly garbage collector to manage objects based on epochs, while NG2C [5–7] divides the heap into many generations to manage objects in finer granularity. ScissorGC [25, 26] improves the performance of old GC for big-data applications. Wang et al. [39] enable elastic memory resource management for multiple JVMs in the cloud environment. Skyway [33] proposes an optimized serialization/deserialization protocol to directly send/receive object graph between JVMs. Taurus [28, 29] coordinates GC in different JVMs to improve the execution time of Spark applications. *GCPersist* also proposes runtime optimizations but focuses on the crash recovery part of big-data processing frameworks.

7 Conclusion

This work proposes *GCPersist*, an easy-to-use NVM programming framework designed especially for resilient applications. *GCPersist* is built atop a novel GC-assisted lazy persistency model, and the evaluation with Intel Optane DC persistent memory shows that *GCPersist* can significantly improve the recovery performance of Spark with trivial overhead during normal execution.

References

[1] Lada A Adamic and Natalie Glance. 2005. The Political Blogosphere and the 2004 US Election: Divided they Blog. In *Proceedings of the 3rd International Workshop on Link Discovery*. ACM, 36–43.

[2] Shoaib Akram, Jennifer Sartor, Kathryn McKinley, and Lieven Eeckhout. 2019. Crystal Gazer: Profile-driven write-rationing garbage collection for hybrid memories. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 1 (2019), 9.

[3] Shoaib Akram, Jennifer B Sartor, Kathryn S McKinley, and Lieven Eeckhout. 2018. Write-rationing garbage collection for hybrid memories. *ACM SIGPLAN Notices* 53, 4 (2018), 62–77.

[4] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. 2016. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 677–694.

[5] Rodrigo Bruno and Paulo Ferreira. 2017. POLM2: automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 147–160.

[6] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 2017. NG2C: pretenuring garbage collection with dynamic generations for HotSpot big data applications. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 2–13.

[7] Rodrigo Bruno, Duarte Patrício, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 28.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[9] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 433–452.

[10] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices* 46, 3 (2011), 105–118.

[11] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 478–493.

[12] Mingkai Dong and Haibo Chen. 2017. Soft Updates Made Simple and Fast on Non-volatile Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 719–731.

[13] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 394–409.

[14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

[15] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: a scalable and efficient persistent transactional memory. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 913–928.

[16] Apache Hadoop. 2009. Hadoop.

[17] Hewlett Packard Enterprise. 2016. Managed Data Structures. https://github.com/HewlettPackard/mds.

[18] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 468–482.

[19] INTEL. 2019. Intel(R) Optane(TM) DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-andtechnology/optane-dc-persistent-memory.html..

[20] INTEL. 2019. Persistent Collections for Java. https://github.com/pmem/pcj.

[21] INTEL. 2020. pmem.io: Persistent Memory Programming. http://pmem.io/.

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[23] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 399–411.

[24] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Statistical Properties of Community Structure in Large Social and Information Networks. In *Proc. Int. World Wide Web Conf.* 695–704.

[25] Haoyu Li, Mingyu Wu, and Haibo Chen. 2018. Analysis and Optimizations of Java Full Garbage Collection. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*. 1–7.

[26] Haoyu Li, Mingyu Wu, Binyu Zang, and Haibo Chen. 2019. ScissorGC: scalable and efficient compaction for Java full garbage collection. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 108–121.

[27] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DUDETM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 329–343.

[28] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *ACM SIGOPS Operating Systems Review* 50, 2 (2016), 457–471.

[29] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[30] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx.. In *EuroSys*. 499–512.

[31] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. 2019. Gerenuk: thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 538–553.

[32] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. 2017. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[33] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting managed heaps in distributed big data systems. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 56–69.

[34] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*.

[35] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 265–276.

[36] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: an easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 316–332.

[37] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 91–104.

[38] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 347–362.

[39] Jingjing Wang and Magdalena Balazinska. 2017. Elastic Memory Management for Cloud Data Analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 745–758.

[40] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing java for more non-volatility with non-volatile memory. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 70–83.

[41] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 478–496.

[42] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 167–181.

[43] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 10–10.